Informatics in Education, 2025, Vol. 24, No. 1, 199–221

DOI: 10.15388/infedu.2025.07

Clusters of Solvers' Behavior Patterns Among Beginners and Non-beginners and Their Changes During an Introductory Programming Course

Heidi TAVETER, Marina LEPP

Institute of Computer Science, University of Tartu, Estonia
e-mail: heidi.taveter@ut.ee, marina.lepp@ut.ee

Received: October 2024

Abstract. Learning programming has become increasingly popular, with learners from diverse backgrounds and experiences requiring different support. Programming-process analysis helps to identify solver types and needs for assistance. The study examined students' behavior patterns in programming among beginners and non-beginners to identify solver types, assess midterm exam scores' differences, and evaluate the types' persistence. Data from Thonny logs were collected during introductory programming exams in 2022, with sample sizes of 301 and 275. Cluster analysis revealed four solver types: many runs and errors, a large proportion of syntax errors, balance in all features, and a late start with executions. Significant score differences were found in the second midterm exam. The late start of executions characterizes one group with lower performance, and types are impersistent during the first programming course. The findings underscore the importance of teaching debugging early and the need to teach how to program using regular executions.

Keywords: behavior features in programming, behavior patterns in programming, programming-process analysis, clustering, introductory programming.

1. Introduction

Analysis of the programming process has become one of the ways to get an overview of how students program and identify what kind of help they need in introductory in programming (Blikstein, 2011; Hosseini *et al.*, 2014; Vihavainen *et al.*, 2014a). For example, some students add code incrementally and test it regularly. However, some write a large part of the code and then start improving it (Hosseini *et al.*, 2014; Meier *et al.*, 2020). The use of trial-error attempts has also been mentioned separately (Blikstein, 2011; Jemmali *et al.*, 2020; Hosseini *et al.*, 2014; Michaeli & Romeike, 2019). The latter has been associated with beginners (Blikstein, 2011). In the context of this article, beginners are those who have never tried to program. In addition, beginners have been

found to use a lot of copying and pasting (Blikstein, 2011; Vihavainen *et al.*, 2014a). It has also been emphasized that students use copying and pasting in the first weeks of the course (Vihavainen *et al.* 2014a), which suggests that there are changes in behavior features in programming during the course. It has also been observed that finding syntax errors is difficult for beginners, especially early in the course (Denny *et al.*, 2012; Marceau *et al.*, 2011). These results show that when studying learner behavior features and patterns in programming, it is necessary to consider previous programming experience and determine how behavior features and patterns in programming change during the course. In this article, a behavior pattern in programming refers to how a group conducts itself throughout the programming process.

2. Literature Review

2.1. *Challenges in Teaching Programming in Introductory Courses*

As the number of people learning programming has grown, this poses several challenges for introductory courses (Luxton-Reilly *et al.*, 2018; Santos *et al.*, 2013; Utting *et al.*, 2013). High dropout and failure rates in introductory programming courses are also continuously problematic (Luxton-Reilly, 2016; Medeiros *et al.*, 2018; Watson & Li, 2014). Therefore, there is a need for further course development, and it is essential to consider and pay more attention to the higher diversity of students (Becker *et al.*, 2019; O'Malley & Aggarwal, 2020; Santos *et al.*, 2013). Students in introductory courses have different levels of previous programming experience, and some do not have any knowledge of the field. Research has shown that students with prior programming experience perform better in introductory courses (Ateeq *et al.*, 2014; Porter & Zingaro, 2014; Veerasamy *et al.*, 2018). It has been emphasized that prior experience influences, particularly, the outcome of the first programming course during the studies (Holden & Weeden, 2003).

From other perspectives, the very beginning of programming education is also essential. Porter and Zingaro (2014) found that the results of weeks three and four are most predictive of final grades. This observation is supported by other works, which also indicate a correlation between the first weeks' results and final grades (Ahadi *et al.*, 2014; Estey & Coady, 2016). Some papers have pointed out that difficulties with syntax errors are also most common at the beginning of the course (Denny *et al.*, 2012; Marceau *et al.*, 2011), and the skills to find and fix syntax errors are related to the final grade (Zhang *et al.*, 2022). The study showed that students who were good at debugging achieved better results on the first exam, and the difference in results increased in the next exam. It aligns with Watson *et al.* (2013), who found that the time spent fixing errors predicts students' performance. Therefore, spotting students who struggle with syntax errors is essential. Smith and Rixner (2019) recommend providing them with targeted instructions. Research has also revealed a need to focus on teaching debugging at the beginning of the course (Zhang *et al.*, 2022). It is a valuable skill that helps fix errors and understand the sequence of the program commands. Teaching debugging is

essential, especially since research has shown that beginners usually do not use debugging (Ardimento *et al.*, 2022; Liu & Paquette, 2023). They also make changes without checking the correctness (Ardimento *et al.*, 2022).

2.2. *Programming-Process Analysis*

Analysis of the programming process has been used in various ways to understand and support programming education. One option is to focus on beginners to understand what differentiates their programming from those who have some previous experience. For example, using an environment that records every keystroke event, it has been found that beginners copy-paste a lot in the beginning when they learn to program (Vihavainen *et al.*, 2014a). This is in line with Blikstein's (2011) work, which was also based on keystroke recording, which found that beginners use more copying, pasting, and customization of the code obtained from external sources. It has also been found that the workload of beginners is significantly higher when solving programming tasks than those with some prior programming experience (Vihavainen *et al.*, 2014b), and they use many trial-and-error attempts (Blikstein, 2011). Another observation has been that students' programming styles vary, and the trial approach characterizes only some of them (Hosseini *et al.*, 2014). Research has shown that some execute the program very late, for the first time, after most of it has been written (Meier *et al.*, 2020). Paying attention to differences in programming styles is essential because students with different programming styles need different support strategies (Blikstein, 2011).

Another important direction in the analysis of the programming process is to identify the students who have difficulties and the characteristics of the programming process that predict higher or lower final exam scores. Based on a snapshot analysis or automatic detection of programming behavior, it was found that weaker students try different options at random without thorough thinking (Heinonen *et al.*, 2014) and are likely to make many submissions in a short time (Bey *et al.*, 2019). At the same time, students with higher results make more changes between submissions (Pereira *et al.*, 2020) and spend more time between compilations (Tabanao *et al.*, 2011). However, another research found that the size of code updates is unrelated to course outcomes (Blikstein *et al.*, 2014). Estey and Coady (2016) revealed that at-risk students are characterized by a combination of low compilation rates and repeated hint usage. Research has also shown that students with better performance deal with errors better (Pereira *et al.*, 2020; Zhang *et al.*, 2022), and the level of debugging skills is the key to good results (Zhang *et al.*, 2022). It is in line with other work based on log data, which revealed that error-resolving time is essential in predicting student outcomes (Watson *et al.*, 2013). Errors as a component related to students' performance have been taken into account in various studies (Bey *et al.*, 2019; Carter *et al.*, 2015; Estey & Coady, 2016; Jadud, 2006; Pereira *et al.*, 2020; Pereira *et al.*, 2021; Price *et al.*, 2020; Watson *et al.*, 2013; Watson & Li, 2014). Attention has also been paid to syntax errors, and it has been found that effective dealing with these is related to better final results (Pereira *et al.*, 2020; Zhang *et al.*, 2022). Fewer errors have also been associated with better

progress (Tabanao *et al.*, 2011). In addition, it has been pointed out that students with better results tend to edit programs with recent runtime errors. Still, students who fail exams deal more with syntax errors or edit programs that have never been executed (Carter & Hundhausen, 2017).

In addition to those mentioned above, further behavior features in programming have been associated with students' performance. Shrestha *et al.* (2022) used keystroke-level analysis. They revealed, focusing on the length of pauses, that the students with fewer medium (3–10 minutes) or long pauses tended to perform better in the exam. More frequent use of medium or long pauses may indicate that students needed to search for additional materials. The greater use of copy-pasting is also associated with lower exam scores (Pereira *et al.*, 2020). It has also been revealed that debugging mode is used most frequently by students with average results (Carter & Hundhausen, 2017). At the same time, students with the best results use the debugger less often, and it is similar to students with worse outcomes. Some studies have focused on changes in student programming patterns throughout the course. For example, it has been found that students who have more changes in the code update patterns (code update pattern means here a combination of code update size and frequency) during the course have better final outcomes (Blikstein *et al.*, 2014). This is supported by Estey and Coady (2016), showing that at-risk students' behavior features in programming appear at the beginning of the semester, and they continue to exhibit the same behavior throughout the semester.

2.3. *Student Profiling*

Researchers have created student profiles based on various programming process characteristics, which describe the different aspects of the process. One way is to use detailed data, including code size increases, decreases, and plateaus. For example, Blikstein (2011) divided students into three main groups based on detailed program process data: Copy and pasters, Mixed-mode, and Self-sufficient. Copy and pasters use existing programs as a starting point, and they have long plateaus as they look for sample programs for copying from instruction materials or their previous work. This coding strategy is characteristic of beginners, and others are more common among non-beginners. Self-sufficient students do not have drastic increases in the number of characters because of copying. They have two main periods combined: linear growth of character count, dealing with errors, and long plateaus without browsing materials. Mixed mode is the combination of the other two. Other research (Hosseini *et al.*, 2014) also used detailed programming data, including increases, decreases, and plateaus in an introductory course, and divided students as follows: Builders, Massagers, Reducers, and Strugglers. Builders work gradually, Massagers do the same but have long plateaus as well, Reducers reduce code already written, and Strugglers struggle to pass any tests. This kind of grouping gives a good overview of students' differences in programming, and analyzing their performance would provide additional essential information.

Some studies use a grouping of students to analyze further how behavior patterns in programming relate to success. In one study, students were divided into three groups

based on the number of unit test runs, and the following groups were discerned: intellects, thinkers, and probers (Sharma *et al.*, 2018). The intellects had the lowest number of runs, the longest time between executions, and the highest number of code changes. The probers had the highest number of runs, the shortest time between executions, and the lowest number of code changes. The research showed that intellects had the highest and the probers the lowest unit test success. We can guess that the probers tend to use a trial-and-error approach. The students with the best results had the lowest number of runs, but it is unclear if some students have more runs but perform well. However, some previous research results are contradicting this. For example, the research that uses code snapshots of every execution and analyzes them with machine-learning techniques shows that the group of students who took consistently smaller steps in moving towards the solution performed better than those who, at some point, suddenly transformed a semi-working program into the correct solution (Piech *et al.*, 2012).

The number of submissions is also a behavior feature used in student profiling. Based on cluster analysis, Bey *et al.* (2019) differentiated three groups. Cluster 1 made more submissions than the others, but Cluster 2 spent more time before submitting and made more changes than the other clusters. Cluster 3 made fewer submissions than Cluster 1, spent less time before submitting, and made fewer changes than Cluster 2. Cluster 3 had the largest number of high performers. It is in line with other research that used cluster analysis based on specific features, which included the number of submissions, the volume of code modifications, and the time between submissions and syntactic correctness (Bey & Champagnat, 2022). They concluded that better-performing students designed the complete solution at the beginning and then submitted and started to fix errors. At the same time, students who wrote only some lines of code and then started compiling had lower final scores in the course. Among others, they found a cluster representing students who are good at designing the global solution but make syntactical errors because they hurry to get an assessment result. Analyzing submissions with other features gives valuable insights into students' behavior patterns but leaves out the process between submissions.

In addition, there are studies where one component of the code construction pattern has been used as the basis for grouping with cluster analysis. For example, a study monitored pauses during the programming process (Shrestha *et al.*, 2022). They analyzed pauses of different lengths during programming, dividing pauses as follows: micro (2–15 seconds), short (16–180 seconds), medium (3–10 minutes), and long (above 10 minutes). Two groups were identified based on pause lengths, concluding that the students who took relatively more micro pauses performed better than those with a higher share of short, medium, and long pauses.

Efforts have also been made to incorporate many different behavior features. A study used large datasets and varied behavior features in programming to look inside the programming process and its context during the course (Pereira *et al.*, 2020). The analysis revealed three clusters based on multiple features. In addition to the usual features related to errors, correctness, use of copy-paste, changes between submissions, etc., they incorporated features that characterize longer processes, for example, IDE (integrated development environment) usage time, number of logins during the four weeks, and

amount of time between the first line of code and assignment deadline. The clusters also differed in the final grade (Pereira *et al.*, 2020). The groups were designated as A, B, and C. A included the most high-performing students, B was in the middle, and students in cluster C performed the worst. The well-performing students have fewer errors, deal with them better, need less time for corrections, make more changes between submissions, and use less copy-pasting. They have also stronger engagement with the course and spend more time in the programming environment.

Some analyses have been done about movements between clusters. For example, researchers analyzed students' movement between the clusters during the three weeks of the course and revealed that low performers had more frequent cluster changes (Bey & Champagnat, 2022). It contradicts other research, which revealed that higher-performing students have more changes in behavior patterns in programming during the course (Blikstein *et al.*, 2014; Estey & Coady, 2016). In conclusion, it can be said that students' behavior features have been used in different ways for profiling. However, some gaps and contradictory results are related to the graduality of creating programs (small steps vs designing the bulk of the solution at first) and changes in behavior patterns.

2.4. *Research Problem*

Dividing students into groups has been employed to identify behavioral patterns in programming that are characteristic of higher-performing students. For instance, a study utilizing cluster analysis to form groups revealed that students with better performance correct errors more quickly, encounter fewer repeated issues with the same errors, and make fewer syntax errors (Pereira *et al.*, 2020). These findings are supported by earlier research conducted by Vihavainen (2013) and Zhang *et al.* (2022). Additionally, higher-performing students tend to copy and paste less frequently, as Pereira *et al.* (2020) highlighted. This observation aligns with the findings of Shrestha *et al.* (2022), another cluster analysis-based study, which demonstrated that students who take fewer long pauses (exceeding 10 minutes) achieve better results. Long pauses were associated with searching for external resources or materials. Furthermore, strong debugging skills have also been identified as a key characteristic of better-performing students (Zhang *et al.*, 2022). There is limited research exploring that debugging skills may also be associated with the timing of the first code execution. Specifically, a study has shown that students who write a substantial portion of their code before the first execution tend to achieve lower exam scores than those who execute their code earlier (Meier & Lepp, 2023). However, whether this finding applies to students in introductory courses with no prior programming experience remains unknown. There have been some attempts to investigate changes in behavior patterns in programming during a course, but the results are conflicting (e.g., Bey *et al.*, 2019; Estey & Coady, 2016). Building on previous research, the question remains: Which behavior patterns in programming characterize beginners, how are these patterns related to performance, and how do these patterns and tendencies differ from those of non-beginners? Focusing on beginners is critical because researchers have pointed to differences in coding between beginners and non-beginners (Blikstein,

2011; Vihavainen *et al.*, 2014b). Furthermore, many challenges beginners face manifest early in the course (Denny *et al.*, 2012; Marceau *et al.*, 2011). Gaining a deeper understanding of beginners' ineffective behavior patterns in programming can help educators focus on improving these skills. It is also essential to monitor changes in thes behavior patterns over the course duration. This provides guidance on the key aspects to emphasize when teaching both at the start of the course and in subsequent stages. Considering the above, the following research questions were posed:

- What types of solvers can be differentiated from the analysis of the programming process (of beginners and non-beginners)?
- Are there statistically significant differences in the midterm exam scores of different solver types (beginners and non-beginners)?
- How persistent are the solver types among beginners and non-beginners over time?

3. Methodology

This study aimed to examine students' behavior patterns in programming among beginners and non-beginners to identify solver types, assess midterm exam scores' differences between solver types, and evaluate the types' persistence. Therefore, cluster analysis, which represents a quantitative approach, was used in this study. A quantitative approach was chosen because it is based on objective measurement and allows the detection of patterns and trends using data and the generalization of findings to a broader population (Dehalwar & Sharma 2024). Clustering can reveal hidden patterns in datasets, and it is common in previous studies to use the k-means algorithm to detect student profiles based on programming-process data (e.g., Bey & Champagnat, 2022; Pereira *et al.*, 2020; Shrestha *et al.*, 2022). Primary data about students' programming process were collected from Thonny logs they submitted in two midterm exams. The Thonny was chosen because it is a Python IDE that logs user actions and generates logs that contain information about students' actions in the IDE during the exam task-solving, for example, runs, error messages, and pastes with timestamps. The sample was formed using a voluntary response sampling strategy (Murairwa, 2015). Specifically, the sample included all students enrolled in the introductory programming course who submitted complete logs and provided responses to the question regarding their programming experience. The course, sample, data collection, and data analysis are described in more detail in the following sections.

3.1. *Overview of the Course*

The study uses data from the fall 2022 course "Computer Programming." The course takes place at the University of Tartu every year and is mainly aimed at students who study Informatics as a major. In addition, there are groups for students with other majors, such as Computer Engineering, Mathematics, Mathematical Statistics, Physics, Chem-

Task 1. Shopping Trip (10 points)

Pille is going shopping. She has prepared a list of groceries she wants to buy. In the file, the information for each grocery item is given in three lines: the first line contains the name, the second line includes the price as a floating-point number, and the third line contains the quantity as an integer. The number of items in the file is unknown. Pille has 15 euros for shopping. She decides to skip buying one grocery item to stay within her budget.

Write a program that:

- Asks the user for the name of the data file.
- Asks the user for the name of the grocery item to skip.
  - Purchases all items if the specified grocery item is not in the file.
- Displays the names of the purchased items on the same line.
- Displays the total cost of the purchase, rounded to two decimal places.
- Indicates whether Pille had enough money to pay for the groceries.
- Write to the file ***money.txt*** how much money Pille had left over or how much was missing. If the amount of money is exact, it should indicate that 0 euros were left over.

Example content of the file:

```
apples
1.3
4
energy drink
2.5
2
ice cream
1.1
7
watermelon
4.0
1
```

Example of program output:

```
Enter the file name: groceries.txt
Which item not to buy: ice cream
Shopping list: apples, energy drink,
watermelon
The total cost is 14.2 euros.
Pille had enough money for shopping!
```

Content of the file *money.txt*:

```
Pille had 0.8 euros left over.
```

Fig. 1. Programming task 1 of the first midterm exam (translated).

istry, and Materials Science. The course is also open to students who want to take it as an elective course. It is a Python course that lasts for 16 weeks and uses a flipped classroom approach, which means that students study course materials about new topics at home before solving programming tasks and answering test questions. After that, they deal with the same topic in a practical session. During the course, they use Python IDE Thonny, designed to learn and teach programming (Annamaa, 2015). Its goal is to offer an IDE that has options that beginners especially need. Among others, its functionality allows users to debug their programs in detail. In addition, it has valuable options for teachers and researchers. It has logging functionality, which saves user actions during the programming process. These actions (with timestamps) include editing the program text whereas pasted text is differentiated from typed text, executing programs, interacting with standard streams (stdin, stdout, stderr), loading and saving files, using stepping commands, etc (Annamaa, 2015). Users can also replay the programming process.

The course includes three exams for testing students' knowledge: two midterm exams and a final exam. The first midterm exam is in the 6th week and covers the following main topics: variables, conditional statements, functions, loops, and basic data exchange with files. The second midterm exam is in the 12th week, focusing on the following main topics: primary data structures as lists, tuples, sets, and dictionaries; nested loops; and data exchange with files. The final exam covers all previous topics, recursion, and introduction to object-oriented programming. The exams consist of two parts: a test focusing on code reading and programming tasks for testing code writing skills. Help resources are not allowed during the test but can be used while solving programming tasks. In the midterm exam, students have 90 minutes to complete both parts, whereas the time limit in the final exam is 180 minutes, and it must be completed in the classroom. Students can get 20 points for programming tasks in each midterm exam. In the final exam, they must solve three programming tasks to get a maximum of 30 points. This study uses data from the second part of the two midterm exams that examine code writing skills. Students solve two larger programming tasks and must use Thonny to do them. They must create two programs based on the instructions. In addition, it is compulsory to submit the logs the environment generates. The reliability of different exam variants was ensured by designing the tasks with the same components, type of input, file content structure, and other requirements, as well as using a shared assessment matrix. An example of the first programming task of the first midterm exam is shown in Fig. 1.

3.2. *Data Collection and Sample*

Primary data were collected from Thonny logs students submitted in both midterm exams. Using a program, the following data were extracted from each student's logs about the programming process and added to a CSV file: starting time, ending time, number of runs, number of error messages, number of syntax errors, number of characters at first run for each program, and number of characters at log submission for each program. Then, it was checked whether the logs contained information about the entire solving time and did not contain information about activities before starting the exam programming tasks. The total complete logs received were 301 for the first and 275 for the second midterm exam. In addition to the log information, each student's midterm exam score was added. Research (O'Malley & Aggarwal, 2020; Vihavainen *et al.*, 2014b) has shown that prior programming experience impacts performance in introductory programming courses, which was also considered. A survey was used at the beginning of the course to determine if they had previous programming experience. This research used one multiple choice question which offered the following options: 1) I have never tried to program, 2) I have tried to program a little but did not make much progress, 3) I can create simple programs, and 4) I have good programming skills, and this course does not give me much new knowledge. For this study, the results were coded according to programming experience in two categories: those with previous experience with programming as non-beginners (students who marked options 2, 3, or 4 in the question) and those with none as beginners (students who marked option 1 in

the question – *I have never tried to program*). There were 79 beginners and 222 non-beginners in the first midterm exam sample and 72 beginners and 203 non-beginners in the second midterm exam sample. The datasets incorporated data on students who had complete logs from both midterm exams to facilitate movement analysis from one cluster to another. There were 233 such students. Of these, 61 were beginners, and 172 were non-beginners.

Four features were included in the analysis using the data collected from logs. The importance of the features is based on previous studies, as referenced in Table 1, which played a crucial role in the selection process. Additionally, during the cluster analysis, more features were included in the testing phase, such as tasks solving time and number of debugging instances. Further details about the process can be found in the data analysis section. The final features were as follows: the number of runs and error messages, the percentage of syntax errors, and the percentage of characters in programs at the first run (the proportion of the total number of characters at the end of task solutions). The percentage of typed characters at the first run is an essential feature in programming not included in previous research that analyzed differences in exam scores of different solver types separately among beginners and non-beginners. This may be a key feature as it indicates the late onset of debugging during programming, which can be related to exam scores. The number of runs and error messages are incorporated because the high number of these is potentially related to the high number of trial-error attempts

Table 1

The description of the features and references to previous studies

Feature	Description	References
Number of runs	The total number of program executions while solving two midterm exam programming tasks.	Hosseini *et al.*, 2014; Jadud, 2006; Meier *et al.*, 2020; Sharma *et al.*, 2018; Tabanao *et al.*, 2011
Number of error messages	The total number of error messages while solving two midterm exam programming tasks.	Carter *et al.*, 2015; Jadud, 2006; Price *et al.*, 2020; Tabanao *et al.*, 2011; Watson *et al.*, 2013
Percentage of typed characters at the first run	The percentage of typed characters at the first run considering two programming tasks.	Late first execution indicates a late start of debugging while programming (Meier & Lepp, 2023). In other previous works, the volume of written code between executions or submissions (Allevato & Edwards, 2010; Bey *et al.*, 2019; Bey & Champagnat, 2022; Pereira *et al.*, 2020; Pereira *et al.*, 2021; Sharma *et al.*, 2018; Zhang *et al.*, 2022) and the executions-editing sequences (Carter & Hundhausen, 2017) have been considered, but not the percentage of typed characters at the first run.
Percentage of syntax errors	The percentage of syntax errors out of the total number of errors that occurred while solving two midterm exam programming tasks.	Use of syntax errors ratio: Bey *et al.*, 2019; Estey & Coady, 2016; Pereira *et al.*, 2019; Pereira *et al.*, 2020; Pereira *et al.*, 2021.

as an ineffective behavior (Sharma *et al.,* 2018). Still, they can also be related to more frequent testing, and it might be ineffective behavior if students do not debug their programs at all (Carter & Hundhausen, 2017). Adding features related to potentially ineffective behavior (many error messages, high percentage of syntax errors) also helps to see which other features are usually accompanied by these behaviors. The descriptions of the features used and references to previous studies are presented in Table 1.

3.3. *Data Analysis*

The data were analyzed using the statistical software IBM SPSS Statistics version 28. The first step was to standardize the scores of features to the same scale. The z-score was used for this purpose. Then, both datasets (one for the first midterm exam and another for the second midterm exam) were divided into two parts based on students' previous programming experience: beginners (students who marked option 1 in the survey – I have never tried to program) and non-beginners (students who marked options 2, 3, or 4 in the survey). Each dataset was analyzed separately using k-means cluster analysis with a maximum of 10 iterations. The purpose was to group students based on features in programming. As k-means cluster analysis typically involves experimenting with different numbers of clusters to determine the most meaningful solution in the context of the research (Jain, 2010), we tried cluster models with three, four, five, six, and seven clusters. It also involved different features in programming beyond the final ones. The analysis showed that the most meaningful solution was with four clusters and features described in Table 1. Each cluster of all datasets (beginners and non-beginners at the first and the second midterm exam) was analyzed to find descriptive statistics of the first and the second midterm exam scores. A Kruskal-Wallis test was conducted to evaluate the existence of any significant differences between clusters. The Mann-Whitney U test was employed to compare different features pairwise. An Alluvial diagram was used to present the moves between clusters.

4. Results

The descriptive statistics of midterm exam scores among beginners and non-beginners are presented in Table 2.

Table 2
Descriptive statistics of midterm exam scores

Programming Experience	Midterm Exam	Students	Mean	Standard Deviation	Min	Max
Beginners	1	79 (26.2%)	11.87	4.77	1	20
	2	72 (26.2%)	14.27	5.58	1	20
Non-beginners	1	222 (73.8%)	15.92	4.33	2	20
	2	203 (73.8%)	17.37	4.00	0	20

4.1. *Solver Types*

To address the first research question, we categorized different types of solvers into distinct clusters. Among beginners and non-beginners, we identified clusters with characteristics similar to the whole group clusters identified in the previous analysis based on the first midterm exam programming process (Meier & Lepp, 2023). For this reason, we used the same names to label the clusters: 1) Frequent pressers of the run button, 2) Receivers of syntax errors, 3) Balanced solvers, and 4) Late starters of the program execution. The beginner and non-beginner clusters for the first and second midterm exams are shown in Fig. 2.

Next, we examine the main differences between the clusters, considering both midterm exams, beginners and non-beginners. The following observations can be made: "Frequent pressers of the run button" differ from others in that they execute programs more than others and receive more error messages (in all cases, $p < 0.001$). "Receivers of syntax errors" can be characterized by the highest ratio of syntax errors (in all cases, $p < 0.05$). In the case of beginners are also characterized by a low number of executions (in all cases except for "Late starters of the program execution" in the first midterm exam, $p < 0.05$). "Balanced solvers" have a below-average rate in all features. The non-beginners among "Balanced solvers" are also characterized by a low ratio of syntax errors (for both midterm exams with "Frequent pressers of the run button" and "Receivers

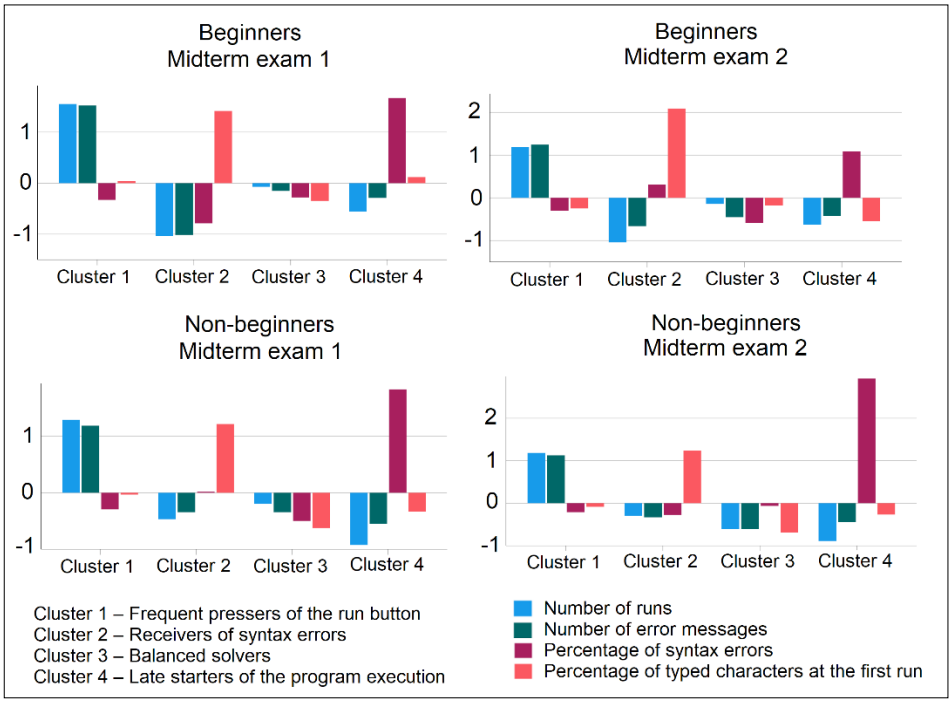

Fig. 2. Clusters at the midterm exams.

of syntax errors" $p < 0.001$). "Late starters of the program execution" had written the highest ratio of program characters by the time of first execution (in all cases, $p < 0.001$). In the case of non-beginners, they are also characterized by a low number of executions (first midterm exam: in all cases, $p < 0.001$; second midterm exam: in all cases except "Balanced solvers," $p < 0.05$).

The distances between cluster centers are different for beginners and non-beginners. For beginners, the most remarkable distance is between "Frequent pressers of the run button" and "Receivers of syntax errors" (the first midterm exam 3.904, the second midterm exam 3.801). The shortest distance is between "Balanced solvers" and "Late starters of the program execution" (the first midterm exam 2.066, the second midterm exam 1.779). For non-beginners, the most remarkable distance is between "Frequent pressers of the run button" and "Late starters of the program execution" (the first midterm exam 3.545, the second midterm exam 4.089), while the shortest distance is between "Receivers of syntax errors" and "Balanced solvers" (the first midterm exam 1.933, the second midterm exam 1.979).

4.2. *Differences in Midterm Exam Scores of Solver Types*

To answer the second research question, we compared midterm exam scores between clusters by separately analyzing beginners' and non-beginners' scores on both midterm exams. The descriptive statistics are presented in Table 3 for beginners and Table 4 for non-beginners. Fig. 3 shows the distributions of the scores by clusters. The Mann-Whitney U test analysis showed that if beginners and non-beginners are considered separately, there are no statistically significant differences between the clusters' first midterm exam scores. However, in the second midterm exam scores, there are statistically significant differences between clusters among both beginners and non-beginners. The second midterm exam score of the beginners who belong to the "Balanced solvers" cluster was statistically significantly higher than that of the clusters "Receivers of syntax errors" ($U = 58$, $p < 0.05$) and "Late starters of the program execution" ($U = 129$, $p < 0.05$). Regarding the non-beginners, however, "Balanced solvers" had a statisti-

Table 3

Descriptive statistics of beginners' midterm exam scores

Cluster	Midterm Exam	Students	Mean	Standard Deviation	Min	Max
Frequent pressers of the run button	1	13 (16.5%)	12.04	3.74	7.5	19
	2	20 (27.8%)	14.68	4.64	7	20
Receivers of syntax errors	1	9 (11.4%)	10.61	6.25	1	20
	2	9 (12.5%)	10.61	6.84	1	20
Balanced solvers	1	43 (54.4%)	12.71	4.54	5	20
	2	26 (36.1%)	16.83	4.19	6.5	20
Late starters of the program execution	1	14 (17.7%)	9.93	5.05	3.5	19.5
	2	17 (23.6%)	11.82	6.09	3.5	20

Table 4

Descriptive statistics of non-beginners' midterm exam scores

Cluster	Midterm Exam	Students	Mean	Standard Deviation	Min	Max
Frequent pressers of the run button	1	54 (24.3%)	15.79	4.11	3.5	20
	2	62 (30.5%)	15.44	4.17	4.5	20
Receivers of syntax errors	1	53 (23.9%)	14.82	4.90	3.5	20
	2	51 (25.1%)	18.51	3.24	3.5	20
Balanced solvers	1	84 (37.8%)	16.55	4.03	4.5	20
	2	79 (38.9%)	18.63	2.77	6.5	20
Late starters of the program execution	1	31 (14%)	16.35	4.29	2	20
	2	11 (5.4%)	14	7.07	0	20

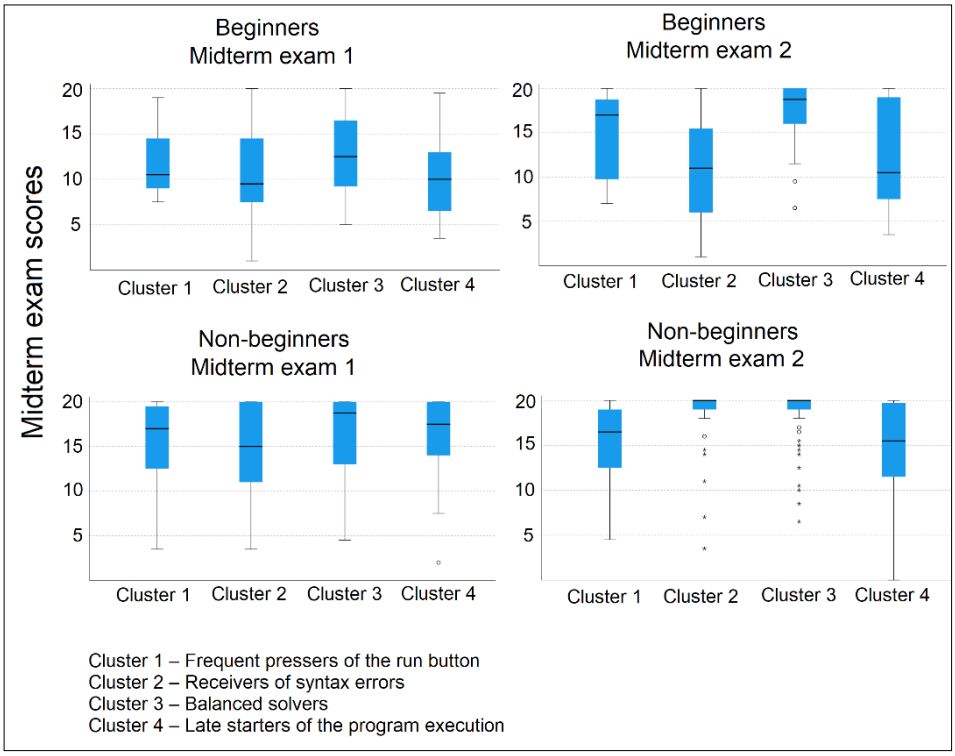

Fig. 3. Distributions of midterm exam scores by clusters.

cally significantly better second midterm exam score than "Frequent pressers of the run button" ($U = 1058$; $p < 0.001$) and "Late starters of the program execution" ($U = 251$; $p < 0.05$). "Receivers of syntax errors" also performed better than "Frequent pressers of the run button" ($U = 698$; $p < 0.001$) and "Late starters of the program execution" ($U = 165$; $p < 0.05$).

4.3. *Persistence of Solver Types*

To answer the third research question, we identified students' transitions from one cluster to another. Two Alluvial diagrams show beginners' and non-beginners' moves between clusters (see Fig. 4 and Fig. 5). The movement of beginners between clusters is shown in Fig. 4. Of 61 beginners, 37 (61%) moved to other groups, while 24 (39%) stayed the same. More details about beginners' moves between clusters are presented in Table 5.

The proportion of moves of non-beginners is similar to that of beginners. The movement of non-beginners between clusters is shown in Fig. 5. Of 172 non-beginners, 101 (59%) moved to other groups, while 71 (41%) stayed in the same one. More details about non-beginners' moves between clusters are presented in Table 6.

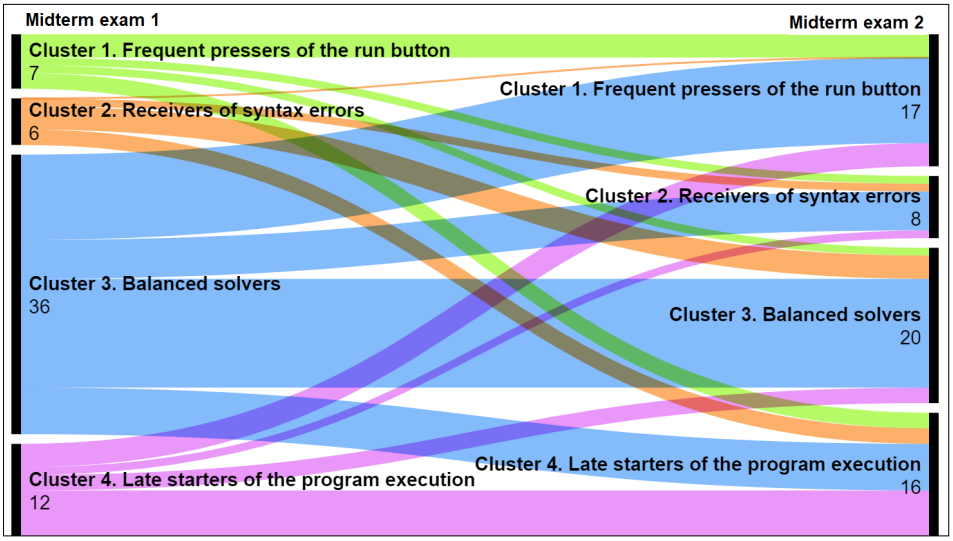

Fig. 4. Beginners' moves from one cluster to another.

Table 5

Beginners' moves between clusters from midterm exam 1 (rows) to midterm exam 2 (columns)

	Cluster 1 (n = 17)	Cluster 2 (n = 8)	Cluster 3 (n = 20)	Cluster 4 (n = 16)
Cluster 1. Frequent pressers of the run button (n = 7)	3 (43%)	1 (14%)	1 (14%)	**2 (29%)**
Cluster 2. Receivers of syntax errors (n = 6)	0 (0%)	1 (17%)	**3 (50%)**	2 (33%)
Cluster 3. Balanced solvers (n = 36)	**11 (31%)**	5 (14%)	14 (39%)	6 (17%)
Cluster 4. Late starters of the program execution (n = 12)	**3 (25%)**	1 (8%)	2 (17%)	6 (50%)

Gray background signifies staying in the same cluster, and numbers in bold have the largest transition probability

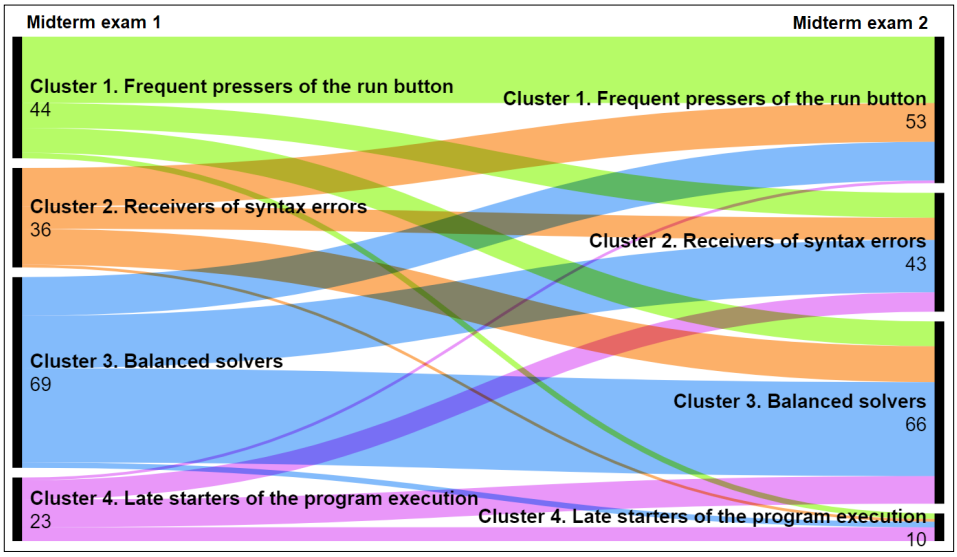

Fig. 5. Non-beginners' moves from one cluster to another.

Table 6

Non-beginners' moves between clusters from midterm exam 1 (rows) to midterm exam 2 (columns)

	Cluster 1 (n = 53)	Cluster 2 (n = 43)	Cluster 3 (n = 66)	Cluster 4 (n = 10)
Cluster 1. Frequent pressers of the run button (n = 44)	24 (55%)	**9 (20%)**	**9 (20%)**	2 (5%)
Cluster 2. Receivers of syntax errors (n = 36)	**14 (39%)**	8 (22%)	13 (36%)	1 (3%)
Cluster 3. Balanced solvers (n = 69)	14 (20%)	**19 (28%)**	34 (49%)	2 (3%)
Cluster 4. Late starters of the program execution (n = 23)	1 (4%)	7 (30%)	**10 (43%)**	5 (22%)

Gray background signifies staying in the same cluster, and numbers in bold have the largest transition probability.

5. Discussion

This study aimed to identify different types of solvers among beginners and non-beginners based on the analysis of the programming process. Solvers can be divided into four groups by behavior patterns in programming based on the following behavior features in programming: the number of runs, the number of error messages, the percentage of characters in programs at the first run, and the percentage of syntax errors. The first group is characterized by many runs and error messages, the second by a large proportion of syntax errors, the third by a balance in all features, and the fourth by a late start with runs. It aligns with previous studies that revealed differences in students' behavior patterns in

programming (Blikstein, 2011; Hosseini *et al.*, 2014; Vihavainen *et al.*, 2014a). The first group that had a large number of runs and error messages probably made a lot of trial-error attempts, as has been mentioned by other authors as well (Blikstein, 2011; Jemmali *et al.*, 2020; Hosseini *et al.*, 2014; Michaeli & Romeike, 2019). The fourth group executed the program for the first time when a larger portion of the program was written. Previous research has also noted that some students add code incrementally, whereas others write a large amount of code and then improve it (Hossein *et al.*, 2014). Late starting with program execution may also be related to using more copying and pasting. Research has shown that beginners who use more copying and pasting might need more code examples, while others need more detailed instructions (Blikstein, 2011). It should be emphasized that studying the different solver types is important because students with different programming styles need different support strategies (Blikstein, 2011).

The characteristics of the groups described above apply to both beginners and non-beginners. Similar overall trends can be explained by the fact that all students studied in the first programming course and were not very experienced. At the same time, there are nuances in which the clusters of beginners differ from non-beginners. In both midterm exams, beginners had, unlike non-beginners, a combination of a higher proportion of syntax errors and a lower amount of runs (second group). Among the non-beginners, a lower amount of runs was accompanied by a late start of the program execution (fourth group). It may indicate that beginners have more trouble with syntax errors, and some try to write programs almost without debugging. Among the non-beginners, a small number of runs is characteristic of those who write a large part of the program before the first execution. It aligns with previous research that has highlighted difficulties with syntax errors as a problem, especially in the first weeks of the course (Denny *et al.*, 2012; Marceau *et al.*, 2011), while it has also been observed that sometimes beginners code without checking if the program is correct (Ardimento *et al.*, 2022).

The study also aimed to investigate if there were differences in both midterm exam scores between different solver types. Unlike in the second midterm exam, the groups had no statistically significant differences in the first midterm exam scores. Among beginners, "Frequent pressers of the run button" and "Balanced solvers" performed better than others, but no statistically significant differences exist. It is possible that students' programming styles are not sufficiently formed at the beginning of the course and do not yet show clear tendencies. Another possible reason is that the tasks are still relatively easy in the first midterm exam, so it is possible to solve the assignments even with inefficient styles. At the same time, there were significant differences in the second midterm exam scores. It can also indicate that skill gaps are increasing during the course. This suggests that those with difficulties need support, especially early in the course. It is supported by Zhang *et al.* (2022), revealing that students with good debugging skills performed better on the first exam, and the difference even increased on the second exam.

The beginners' and non-beginners' results on the second midterm exam show that the late start of the program execution characterizes one group of the students whose midterm exam scores are lower. In addition, both the high proportion of syntax errors and the late start of the program execution combined with a small number of runs are patterns

that tend to be related to lower performance. This is essential knowledge, mainly because weaker students have often been associated with a large number of trial-error-attempts (Bey & Champagnat, 2022; Heinonen *et al.*, 2014; Hosseini *et al.*, 2014; Pereira *et al.*, 2020; Tabanao *et al.*, 2011), but there are other kinds of weaker students as well who do not use program execution enough. Also, research has found that students who fail exams struggle with editing programs that have never been executed (Carter & Hundhausen, 2017). This is primarily an indication of insufficient debugging skills. Some students try to write as much of the program as possible without running it or using a debugger if necessary. This finding shows the need to pay more attention to program debugging at the beginning of the course and use teaching techniques that help improve these skills. To suggest to run the program when some parts are written is also helpful. Teaching assistants can also demonstrate how to write programs using executions regularly during the programming process. It is an essential focus in the first weeks, which helps find mistakes more efficiently and improves the understanding of how programs work.

Interestingly, for beginners, a higher proportion of syntax errors is correlated with worse performance, whereas for non-beginners it characterizes one of the better-performing groups. This result needs further investigation, but it indicates that not only the proportion of syntax errors but also the error correction skills and speed play a role (Pereira *et al.*, 2020; Watson *et al.*, 2013; Zhang *et al.*, 2022). So, it is possible that the group of non-beginners who receive proportionally more syntax errors may be able to correct them quickly. Bey and Champagnat (2022) identified a group of students who were good at designing solutions but made syntactical errors because of hurrying. The fact that two distinct groups of non-beginners achieve significantly better results than the two other groups shows that no single pattern characterizes successful students. In teaching, it is also necessary to consider that students can work effectively in different ways.

The study also aimed to determine how persistent the solver types are over time among beginners and non-beginners. The results show that groups are not persistent during the first programming course – more students moved to another group than stayed in the same one, both among beginners and non-beginners. The reasons why students change groups need further research. Further investigation is also required to determine which students stay in the same group and which change groups and how it is related to course outcomes. Previous research with different features has shown that changes over time indicate better results (Blikstein *et al.*, 2014). At the same time, a study based on other various features concluded that those who perform worse change the cluster (Bey *et al.*, 2019).

In conclusion, it is essential knowledge that the late start of the program execution combined with a small number of runs are patterns that tend to be related to lower performance. It is crucial from this point of view that lower performance is often associated with many trial-error-attempts (Bey & Champagnat, 2022; Pereira *et al.*, 2020). This finding shows the need to focus on lower-performing students who don't debug their programs during the writing process enough or not at all. The limited use of debugging is also mentioned in other research (Ardimento *et al.*, 2022; Liu & Paquette, 2023). This paper also adds knowledge that solver types are not persistent during the first programming course among beginners and non-beginners alike.

6. Conclusion

Our research demonstrated that solvers can be divided into groups based on behavioral patterns, which are similar among beginners and non-beginners. We also found patterns that indicate better performance and observed that some groups can perform equally well despite differences, which shows diversity. It was also revealed that solver types were not persistent during the first programming course. An important result was detecting some characteristics that tend to be related to lower performance. In particular, we would like to point out a newly found feature – the late start of the program execution, which is related to lower outcomes among beginners and non-beginners. It is well-known that lower outcomes can often be related to the number of errors. The research now revealed another type of low-performer besides error-related patterns. Answers to the research questions are presented in sections 4.1, 4.2, and 4.3.

These results help teachers to consider specific details and diversity in students' styles. In the context of the result of this study, an essential recommendation is to teach how to program in such a way that executing the programs is a natural part of the programming process. Information about different behavior patterns also gives teachers helpful information for proposing and using different support strategies. Namely, it can sometimes be helpful to pay attention to the differences in solver types generally in practical sessions. Teaching assistants can also explain to students that those who execute programs the first time when a large part of the program is written tend to get lower grades. It is also possible to give different exercises to those who struggle with syntax errors or never execute programs. Some beginners, however, need more examples, and others need more detailed instructions. In addition, the revealed patterns are helpful for students to reflect on their programming styles and get valuable information, which helps them notice their behavior patterns and improve their skills. The findings also show the need to pay more attention to debugging in the first weeks of the course. It helps decrease the number of students who try to program without executing or leave it quite late to run programs for the first time. In addition, using debuggers incorporated into programming environments is also very helpful for students in finding mistakes. These kinds of tools are not only helpful for debugging itself but also for learning how programs work and in which order commands are executed. Introducing and encouraging the use of these kinds of tools is especially essential in large courses because these enable students to be more thorough and independent as they work with programs and engage with their mistakes. It would also be helpful if teaching assistants focused more on explaining in the first few weeks that even short programs consist of parts. Maybe offer exercises that require writing programs incrementally. They can also teach by demonstrating how to write programs using regular executions.

It is also necessary to point out some limitations. The data were collected in one university from a course where most students study Informatics as a major. Although there were groups from other major fields, the results may somewhat differ with a broader diversity of fields. Obtaining information from logs allows detailed information about activities in the programming environment but not outside. To get a broader perspective in the future, it may be helpful to incorporate additional information from other

environments they use during the course, such as the learning management system. As future work, the persistence of solver types and their correlation to course outcomes certainly needs a more detailed investigation with a larger sample. It is essential to map the tendencies that are more or less efficient during the course. It would be important to identify the types of moves between groups associated with higher and lower exam scores and develop practical applications of this knowledge to improve teaching. It is also not clear whether solver types are impersistent only during the first programming course or during a longer period as well. Another possible direction for future research is to examine the lower-performing solver types more closely.

Acknowledgments

This work was supported by the Estonian Research Council grant "Developing human-centric digital solutions" (TEM-TA120).

References

Ahadi, A., Lister, R., & Teague, D. (2014). Falling Behind Early and Staying Behind When Learning to Program. In *PPIG* (14). 2014_06_PPIG_AhadiListerAndTeague_EarlyIndicatorsOfSuccess_May30-Time2200_PDF

Allevato, A., & Edwards, S. H. (2010). Discovering patterns in student activity on programming assignments. In *ASEE Southeastern Section Annual Conference and Meeting*. Virginia Polytech. Inst. and State Univ. Blacksburg, Virginia, 2010. http://se.asee.org/proceedings/ASEE2010/Papers/PR2010All158.PDF

Annamaa, A. (2015). Introducing Thonny, a Python IDE for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM. 117–121. https://doi.org/10.1145/2828959.2828969

Ardimento, P., Bernardi, M. L., Cimitile, M., Redavid, D., & Ferilli, S. (2022). Understanding Coding Behavior: An Incremental Process Mining Approach. *Electronics*, 11(3), 389. https://doi.org/10.3390/electronics11030389

Ateeq, M., Habib, H., Umer, A., & Rehman, M. U. (2014). C++ or python? which one to begin with: A learner's perspective. In *2014 International Conference on Teaching and Learning in Computing and Engineering*. IEEE. 64–69. https://doi.org/10.1109/latice.2014.20

Becker, B. A., & Quille, K. (2019). 50 years of CS1 at SIGCSE: A review of the evolution of introductory programming education research. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM. 338–344. https://doi.org/10.1145/3287324.3287432

Bey, A., Pérez-Sanagustín, M., & Broisin, J. (2019). Unsupervised automatic detection of learners' programming behavior. In *European Conference on Technology Enhanced Learning*. Springer, Cham. 69–82. https://doi.org/10.1007/978-3-030-29736-7_6

Bey, A., & Champagnat, R. (2022). Analyzing Student Programming Paths using Clustering and Process Mining. In *Proceedings of the 14th International Conference on Computer Supported Education*. Scitepress. 76–84. https://doi.org/10.5220/0011077300003182

Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*. ACM. 110–116, https://doi.org/10.1145/2090116.2090132

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23(4), 561–599. https://doi.org/10.1080/10508406.2014.954750

Carter, A. S., Hundhausen, C. D., & Adesope, O. (2015). The normalized programming state model: Predicting

student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. 141–150. https://doi.org/10.1145/2787622.2787710

Carter, A. S., & Hundhausen, C. D. (2017). Using programming process data to detect differences in students' patterns of programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM. 105–110. https://doi.org/10.1145/3017680.3017785

Dehalwar, K., & Sharma, S. N. (2024). Exploring the Distinctions between Quantitative and Qualitative Research Methods. *Think India Journal*, *27*(1), 7–15.

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012). All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education.* ACM. 75–80. https://doi.org/10.1145/2325296.2325318

Estey, A., & Coady, Y. (2016). Can interaction patterns with supplemental study tools predict outcomes in CS1? In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 236–241. https://doi.org/10.1145/2899415.2899428

Holden, E., & Weeden, E. (2003). The impact of prior experience in an information technology programming course sequence. In *Proceedings of the 4th Conference on Information Technology Curriculum*. ACM. 41–46. https://doi.org/10.1145/947121.947131

Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014). Exploring problem solving paths in a Java programming course. In *Proceedings of Psychology of Programming Interest Group Annual Conference,* Brighton, UK, 25–27 June 2014. 65–76.

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*. ACM. 73–84. https://doi.org/10.1145/1151588.1151600

Jain, A. K. (2010). Data clustering: 50 years beyond K-means. Pattern Recognition Letters, 31(8), 651–666. https://doi.org/10.1016/j.patrec.2009.09.011

Jemmali, C., Kleinman, E., Bunian, S., Almeda, M. V., Rowe, E., & El-Nasr, M. S. (2020). MAADS: Mixed-methods approach for the analysis of debugging sequences of beginner programmers. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Portland, OR, USA. 86–92. https://doi.org/10.1145/3328778.3366824

Liu, Q., & Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies. In *LAK23: 13th International Learning Analytics and Knowledge Conference*. ACM. 637–643. https://doi.org/10.1145/3576050.3576094

Luxton-Reilly, A. (2016). Learning to program is easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 284–289. https://doi.org/10.1145/2899415.2899432

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A. N., Ott, L., ... & Szabo, C. (2018). Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education,* ITICSE. ACM. 55–106. https://doi.org/10.1145/3293881.3295779

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education.* ACM. 499–504. https://doi.org/10.1145/1953163.1953308

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, *62*(2), 77–90. https://doi.org/10.1109/te.2018.2864133

Meier, H., Tõnisson, E., Lepp, M., & Luik, P. (2020). Behaviour patterns of learners while solving a programming task: An analysis of log files. In *IEEE global engineering education conference (EDUCON)*. IEEE. 685–690. https://doi.org/10.1109/educon45650.2020.9125134

Meier, H., & Lepp, M. (2023). Clusters of Solvers' Behavioral Patterns Based on Analysis of the Programming Process. In *2023 IEEE Frontiers in Education Conference (FIE)*. IEEE. 1–6. https://doi.org/10.1109/fie58773.2023.10343479

Michaeli, T., & Romeike, R. (2019). Current status and perspectives of debugging in the k12 classroom: A qualitative study. In *2019 IEEE Global Engineering Education Conference (EDUCON)*. IEEE. 1030–1038. https://doi.org/10.1109/educon.2019.8725282

Murairwa, S. (2015). Voluntary sampling design. *International Journal of Advanced Research in Management and Social Sciences*, 4(2), 185–200.

O'Malley, C., & Aggarwal, A. (2020). Evaluating the Use and Effectiveness of Ungraded Practice Problems in an Introductory Programming Course. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. ACM. 177–184. https://doi.org/10.1145/3373165.3373185

Pereira, F. D., Oliveira, E. H., Fernandes, D., & Cristea, A. (2019). Early performance prediction for CS1 course students using a combination of machine learning and an evolutionary algorithm. In *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*. 2161, 183–184. IEEE. https://doi.org/10.1109/icalt.2019.00066

Pereira, F. D., Oliveira, E. H., Oliveira, D. B., Cristea, A. I., Carvalho, L. S., Fonseca, S. C., ... & Isotani, S. (2020). Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming. *British journal of educational technology*, *51*(4), 955–972. https://doi.org/10.1111/bjet.12953

Pereira, F. D., Fonseca, S. C., Oliveira, E. H., Cristea, A. I., Bellhäuser, H., Rodrigues, L., ... & Carvalho, L. S. (2021). Explaining Individual and Collective Programming Students' Behavior by Interpreting a Black-Box Predictive Model. *IEEE Access*, *9*, 117097–117119. https://doi.org/10.1109/access.2021.3105956

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM. 153–160. https://doi.org/10.1145/2157136.2157182

Porter, L., & Zingaro, D. (2014). Importance of early performance in CS1: two conflicting assessment stories. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. ACM. 295–300. https://doi.org/10.1145/2538862.2538912

Price, T. W., Hovemeyer, D., Rivers, K., Gao, G., Bart, A. C., Kazerouni, A. M., ... & Babcock, D. (2020). Progsnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 356–362. https://doi.org/10.1145/3341525.3387373

Santos, Á., Gomes, A., & Mendes, A. (2013). A taxonomy of exercises to support individual learning paths in initial programming learning. In *2013 IEEE Frontiers in Education Conference (FIE)*. IEEE. 87–93. https://doi.org/10.1109/fie.2013.6684794

Sharma, K., Mangaroska, K., Trætteberg, H., Lee-Cultura, S., & Giannakos, M. (2018). Evidence for programming strategies in university coding exercises. In *Lifelong Technology-Enhanced Learning: 13th European Conference on Technology Enhanced Learning, EC-TEL 2018, Leeds, UK, September 3–5, 2018, Proceedings 13*. Springer International Publishing. 326–339. https://doi.org/10.1007/978-3-319-98572-5_25

Shrestha, R., Leinonen, J., Zavgorodniaia, A., Hellas, A., & Edwards, J. (2022). Pausing While Programming: Insights From Keystroke Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE. 187–198. https://doi.org/10.1109/icse-seet55299.2022.9794163

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research*. ACM. 85–92. https://doi.org/10.1145/2016911.2016930

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., ... & Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*. ACM. 15–32. https://doi.org/10.1145/2543882.2543884

Vihavainen, A. (2013). Predicting students' performance in an introductory programming course using data from students' own programming process. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. IEEE. 498–499. https://doi.org/10.1109/icalt.2013.161

Vihavainen, A., Helminen, J., & Ihantola, P. (2014a). How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. ACM. 109–116. https://doi.org/10.1145/2674683.2674692

Vihavainen, A., Luukkainen, M., & Ihantola, P. (2014b). Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information Technology Education*. ACM. 21–26. https://doi.org/10.1145/2656450.2656473

Watson, C., Li, F. W., & Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. IEEE. 319–323. https://doi.org/10.1109/icalt.2013.99

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ACM. 39–44. https://doi.org/10.1145/2591708.2591749

Zhang, Y., Paquette, L., Pinto, J. D., Liu, Q., & Fan, A. X. (2023). Combining latent profile analysis and programming traces to understand novices' differences in debugging. *Education and Information Technologies*, 28(4), 4673–4701. https://doi.org/10.1007/s10639-022-11343-7

H. Taveter is a junior lecturer in informatics and a doctoral student at the Institute of Computer Science at the University of Tartu. Her research focuses on didactics of programming, particularly analyzing the learning and teaching of programming by investigating the programming process.

M. Lepp is an associate professor in informatics and the head of the chair of programming languages and systems at the Institute of Computer Science at the University of Tartu. Her current research interests are focused on AI in programming education, didactics of programming, programming MOOCs, and assessment with computers.